# Chess Game Final Design

By: Ruozhen Gu( r8gu), Yining Song(ynsong) and Zhengchun Wang(z383wang)

## 1. Overview

While we keep the same logics throughout the whole game implementation, our final and original UML may vary in many different ways. For original UML, we have created basic classes to perform most rules of chess game. Neither are those class hierarchy nor the design methodologies changed in any major way between due data 1 and 2. However, we have decided to add multiple functions, each serving a specific goal to improve both class styling and cohesions. For example, originally we are trying to implement moving, castling and En Passant in main function. But doing so will result in a giant main function with lots of redundant codes. Having a corresponding function for each functionality will make our codes more concise and clear. In addition, to perform graphical display, we have to add draw functions for every subclasses of pieces and some supplemental fields are added in player class just to make our class more flexible in chess rules execution.

It is also worth mentioning that “add_chess”, “remove_chess” and two different “init” functions are added to give the user capacity of setting up his own board. We have made the whole program more robust by tolerating user's wrong input and more interactive by printing real-time information of player’s status and error message to improve the game experience. Those are the most fundamental changes we have made to UML and functions contained in it.

Nothing dramatic is changed for our plan of attack for the past two weeks. Our team is able to divide reasonable amount of workload to each teammate and the whole implementation is able to finish within 2/3 of the time as we planned. Every day, we plan to have a 15 min Skype meeting so that each individual could have a good understanding of current process and ensure the compatibility of each other's implementation. As each person has a specific focus and is responsible of compiling his or her own class files, we are eventually able to compile the whole program with little syntax errors and bugs.

## 2. Important features of implementation

### 2.1 An abstract class "Piece"

Rather than controlling a single role, chess games actually allows users to make attacking strategy using all six different pieces of chess with their associated attacking pattern. That means for each function we wrote, we have to guarantee it can be applied on different roles. Since those pieces may only differ in the way of attacking but have all other feature the same (like the coordinator to show the position). Therefore, we decide to create an abstract class "Piece" which contains the necessary fields and six subclasses will be inheriting from "Piece". One of the largest benefits of having this abstract class is that when we attach chess into "player", we do not have to specify what type of chess it is but can just name it "piece". Since all subclasses inheriting the fields from mother class, it also reduces the human work of including the same fields into all six classes.

## 2.2 Fields and methods inside "Piece"

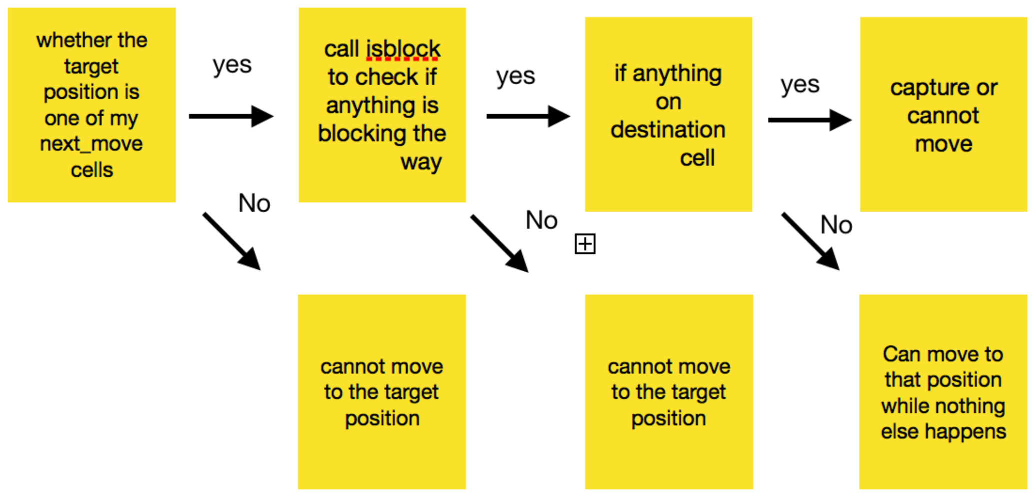
Using the strategy of an abstract class can be significantly beneficial to our role implementation. However, the next big question is, what fields or methods should we put them into this abstract class so that all subclasses will have our desired functionalities?

One of the most important features for implement chess rules is to check the existence of a chess on the board. It is impossible to know when a capturing should be performed if we cannot even tell the status of destination cell. Similarly, we do not know if the movement of a chess will be blocked or not without the information of each cell on its path. Therefore, finding a quick way of checking the status of a cell should be one of the most foundational problem to solve.

At the beginning, we are trying to define two player classes and each one holds a vector of all its alive pieces. Every time we need to check the status of a cell, we can simply go through all those vector of pieces and using getter function to achieve the x and y coordinator values. By comparing if any chess role has desired (x,y) location, we can know whether there exists a piece at that location correspondingly. At the same time, we also have to look through the other player's vector of pieces and perform the same checking procedure since the other player may have a piece on that location as well, which might be captured by us.

The above method seems viable but inefficient to some extent. Performing status checking could be something happening relatively often and we should be able to find an alternative way to streamline the whole process. That's how we come up with a brand new idea to check the status inside "Board": eventually, to display the board, we will have to create a class board which contains all the 64 cells on the board. We display the identity of each cell (whether it's a black or white chess and what's its role) by calling the getter function inside "Cell" class. So maybe we can just add one or two fields inside class "Cell" which indicates: 1. whether this is a white or black piece. 2. which role is this cell currently holding. Thus, every time when we need to check the status of a position, we can easily call getter function in board to get that corresponding cell and then apply getter function again in "Cell" to check its status information. So to fulfill this, we just need to do 3 changes: 1. add fields inside "Cell" for status indication. 2. add a pointer to the board inside class "player" and 3. every time we move any piece, make sure we update the corresponding cell in "Board" as well.

Next, let's consider how to implementation the attacking function. even though the word "attacking" sounds like something complex but it is actually just the possible movements of a piece of chess. As long as the opponent's chess is one of my available next movement and nothing is blocking in between, we can perform capturing directly (make sure you do not attack king). that means if we could have a field for each chess to recording the next available movements based on its current position and update it every time after its movement. Then we can just decide whether a chess can be captured by checking if the position values are inside that field. And that's exactly why we have a function named "produce_nextmove" which is to update the next move filed (vector of vector of int). the function will be called whenever the x and y values are modified for current chess. However, before we actually perform the capturing, we do have to check if there is any chess in between the target position and my position. we implement the function "isblock" which consumes the target position int x and int y and the function will first filtering on all the cells that are between target position and my position and then use the board pointer to check the status of each of those cells.  To visualize the process, please check the graph below:


whether the target position is one of my next_move cells
yes
No
call isblock to check if anything is blocking the way
yes
No
if anything on destination cell
yes
No
capture or cannot move
cannot move to the target position
cannot move to the target position
Can move to that position while nothing else happens

## 2.3 Computer player

Unlike other simple chess game, we will need to implement a simple AI player to compete with human players. Before going through the actual implementation, we first defined four levels of difficulties which user can specify at the beginning of the game.

Level 1:

This is the easiest level that our AI will just perform a random valid move. To implement this, we will just loop through all the available chess and try to perform a random movement as long as it is valid.

Level 2:

Rather than performing a random move, the AI player will actually check which available chess can perform a capturing or check movement. We can realize that by checking each next_move of each chess to see which one can create a check or capturing conditions while not putting our king in check. If neither of those two movements are available, then we will simply perform a random valid move.

Level 3:

This is an advanced level of level 2. Not only do we need to check for available capturing/checking moves, but also we will try to avoid a capturing first. And we should give the priority to avoiding a capturing case. Similarly, we loop through the opponent's possible chess and try to see if any of our chess is in their next_move or not. if so, we will try to perform any available next move of our chess to avoid.

Level 4:

For level four, it will include all the featured within level 2 and 3 but also try to perform a relatively most aggressive move. By saying "relatively most aggressive", we are assigning each chess role an integer to indicate its aggressive level. As generally known, a queen is always more aggressive than a rook as it can move in literally any of the directions with any of the steps. Similarly, as the key of this game is to protect of king, we will assign "King" with the lowest aggressive level so that it won't try to put itself in check condition by consistently moving around. (But if our king is in check, we will try to move king or perform castings or use other chess to block the attacking route of opponent first). In addition, we will teach our AI some attacking strategies and make sure AI can react to any potential attacking of opponent which may later put our king at check condition.

## 2.4 Observer Pattern

We applied observer pattern for text displaying the game. We have “board” class as subject and “Piece” class will be observed. Remember that “Piece” is just an abstract class that is used to build all the actually chess roles occurring on the board and we use the specific getter functions inside “Cell” to retrieve the information of each cell position. (chess role and colour). We attach “testdisplay” pointer within all the cells which will be later used to print the cell information.

## 3. Implementation within each class

Now let’s check what fields and methods are actually in each class and how are they functioning to create this chess game.

## 3.1 Piece:

Piece is a pure virtual class used to create different actual chess player:

***Please note the bold functions are new features added into UML***

Methods:

+ SetX(int x) : void : setter function to modify value of current x-axis for each chess

+ SetY(int y) : void : setter function to modify value of current  y-axis for each chess

+ SetX_prev(int x) : void : setter function to modify x_prev which is the x-axis of last move

+ SetY_prev(int y) : void : setter function to modify x_prev which is the x-axis of last move

+ SetAlive() : void : if the chess is alive=true, calling this function will make it alive=false

+ getX() : int : getter function to get the information about current x-axis for each chess

+ getY() : int : getter function to get the information about current y-axis for each chess

+ getX_prev() : int : getter function to get the previous x-axis for each chess

+ getY_prev() : int : getter function to get the previous x-axis for each chess

**+ getNext_move() : std::vector<std::vector<int>> &** : get available next move vector reference so that we can directly mutate it as well

+ getAlive() : bool : getter to see if this chess is still alive or not

+ virtual produce_next() : void : pure virtual method to produce the available next move based on current position of each chess

+ attack(int x, int y) : bool : check if (x,y) is inside next_move. if so, it will be attacked, return true. it returns false otherwise

**+ get_name() : char :** getter to check what role the current chess is

**+ virtual isblock(int r, int c) :** bool : pure virtual method to see if there's piece between(r,c) and (x,y)

## 3.2 King:

King inherits from Piece and can move in any direction for one step. There is only one King for each player and each player can checkmate King to win the game.

***Please note the bold functions are new features added into UML***

Methods:

+ produce_next() : void : produce next available moves from 8 directions as vector<int> format.

**+ isblock(int r, int c)** : bool : try to check if there is any piece between (r,c) and current (x,y)

## 3.3 Queen:

Queen inherits from Piece and can move in any direction for whatever steps. There is only one Queens for each player.

***Please note the bold functions are new features added into UML***

Methods:

+ produce_next() : void : produce next available moves from 8 directions as vector<int> format.

**+ isblock(int r, int c)** : bool : try to check if there is any piece between (r,c) and current (x,y)

## 3.4 Bishop:

Bishop inherits from Piece and can move diagonally for whatever steps. There are two Bishops for each player

***Please note the bold functions are new features added into UML***

Methods:

+ produce_next() : void : produce next diagonally available moves as vector<int> format.

+ **isblock(int r, int c)** : bool : try to check if there is any piece between (r,c) and current (x,y)

## 3.5 Rook:

Rook inherits from Piece and can move horizontally or vertically for whatever steps. There are two Rooks for each player.

***<u>Please note the bold functions are new features added into UML</u>***

Methods:

+ produce_next() : void : produce next horizontally and vertically available moves

+ **isblock(int r, int c)** : bool : try to check if there is any piece between (r,c) and current (x,y)

## 3.6 Knight:

Knight inherits from Piece and can displace itself 3 units in any combo of left, right, up or down directions. It can also jump over pieces. There are two Knight for each player

***<u>Please note the bold functions are new features added into UML</u>***

Methods:

+ produce_next() : void : produce next horizontally and vertically available moves

+ **isblock(int r, int c)** : bool : always produce false as it will never be blocked

## 3.7 Pawn:

Pawn inherits from Piece and can move one step up or down depending on the different sides of players. Pawn can move two steps forward on its first move and one step for the following moves. Pawn can promote to whatever chess once it reaches the other side. there are 8 Pawn for each player.

***Please note the bold functions are new features added into UML***

Methods:

+ produce_next() : void : produce next available moves from up direction

+ **isblock(int r, int c)** : bool : try to check if there is any piece between (r,c) and current (x,y)

## 3.8 Player

A game has two players. A player can either be human or Computer, in either case, main function will call the corresponding move function to move a piece. Player has important features including checking whether the game is in checkmate, check or stalemate condition. It will also be responsible for mutating the corresponding cell on board using board pointer for text displaying purpose.

***Please note the bold functions are new features added into UML***

Methods:

+ set_player(Player *p2) : void : define whether player is human or computer using level

+ reduce_numAlive() : void : reduce number of alive pieces by 1

+ get_numAlive() : int : getter to see how many pieces are alive for this player

+ **get_chess() : std::vector<Piece*> &** : get alive cheeses reference so we can modify it

+ set_level(int level) : void : user can specify the level for computer player

**+ set_default() : void :** attach on the default pieces into chess

**+ add_chess(char c, int x, int y) :** void : user can set up the board by adding specific chess

**+ remove_chess(int x, int y) : void :** user can set up the board by removing specific chess

+ **capture(int x, int y) :** when a piece is captured, call this to remove the chess that is captured and mutate board accordingly

+ **castling(int new_x, int new_y, bool *s) : void :** check if castling conditions are met and perform castling for king and rook. It will mutate board accordingly

+ **human_move(char promote = 'z', int x, int y, int x_new, int y_new, bool *success) : void :** this function will examine the human input to check whether a valid move has been request by human player. If so, what actions should we execute and check if a checkmate, check or stalemate condition is met after the execution. The function mutates board accordingly for the same changes

+ **move_random_piece() : void :** randomly perform a valid move on any available piece and check for specific conditions (i.e.checkmate) after the movement. Board may be mutated accordingly

+ **try_attack() : void :** function to try to perform an attacking movement or try to put the other king at check condition. If no attacking or check conditions can be executed, move a random piece. Board will be changed accordingly

+ **predict_capture(int x, int y) : bool :** if position x y will be captured by any opponent chess

+ **computer_move() : void :** based on different levels, perform a movement as computer player

+ **isCheck() :** bool : call this function to see if my king is at check condition

+ **isCheck2(int x, int y) :** check if we move king to position (x,y), will king be in check

+ **checkAfterMove(Piece *p) : bool** : check if our king will be in check condition if we move a

+ isCheckmate() : bool : check if my king is at checkmate condition in which our king cannot make any further moves since those will all be attacked

+ **ifCastling(Piece *k, Piece *r) : bool :** check if all conditions for king and rook are met so that we can perform castling

+ isStalemate() : bool : check if it is a stalemate condition in which no player can make any further moves. If so, print the information and end the program

## 3.9 score

Score is a class created in main function so that we can easily trace the score for white and black players. A checkmate or resign will add 1 point to the other player. A draw in stalemate condition will add 0.5 points to both player.

***Please note the bold functions are new features added into UML***

Method:

+ getWhite() : double : getter to check the final score of the white player

+ getBlack() : double : getter to check the final score of the black player

+ setWhite(double score) : void : add score to white player

+ setBlack(double score) : void : add score to black player

+ **printScore() : std::string:** when all games finish, print out the final result accordingly

## 3.10 cell

Cell serves as a holder for coordinator information. It also tells whether there is a piece placed at this location. If so, is it belonging to white player or black player. Later those information will be notified to display

***Please note the bold functions are new features added into UML***

+ get_occupied() : char: check if there is any piece at this location. If so, what is that piece

**+ set_occupied(char occ) : void :** set occ to show a piece placed/removed at this location

+ get_x() : int

+ get_y() : int

+ notifyDisplay(TextDisplay* td) : pass the information of each cell for text displaying

## 3.11 board

Board can initialize the entire board based on user's preference. It contains a 2D array of cells and can mutate the cell inside for displaying

***Please note the bold functions are new features added into UML***

Methods:

+ get_cell(int x, int y) : Cell * : achieve a pointer to a cell so that we can later modify its status

+ init() : void : clear the board and setup the board as default

**+ place(char name, int x, int y) : void :** place a new cell inside the 2D array

**+ remove(int x, int y) :** void : remove a existing cell inside the 2D array

+ check_occupied(int x, int y) : bool : check the status of a specific cell at (x,y)

**+ init_setup() : void :** init an empty board with 2D array.

friend std::ostream & operator<<(std::ostream &out, const Board &g) : calling the cout overloadding function in text display

## 3.12 textdisplay

Textdisplay is used to display the board and each chess piece based on requirement

***Please note the bold functions are new features added into UML***

+ notify(char cell, int x, int y) : void : print either white or black chess piece on board

friend std::ostream &operator<<(std::ostream &out, const TextDisplay &td) : print the cell based on its status and also frame of the board

## 3.13 graphicaldisplay

Graphicaldisplay is used to display the board and each chess piece based on requirement using colorful graphics

***Please note the bold functions are new features added into UML***

**+ notify(char cell, int x, int y) : void : print either white or black chess piece on board**

## 4. Questions and answers

*Question 1. Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.*

We will create an array of array, and each array in the array contains series of positions which stands for standard openings such as Giuoco Piano, King's Gambit, Sicilian Defense, etc.(htt ps://www.dwheeler.com/chess - openings/) For example, Giuoco Piano starts with: 1. e4 e5 2. Nf3

Nc6 3. Bc4 Bc5. When one of the player is computer, the computer will go through this array, randomly pick a standard opening and stick with it. As for response to opponent’s moves, every time when the computer wants to move, it will first use the level 3 prefer avoids algorithm to avoid potential attack, then if there is no potential attack, it will continue move regarding with the standard opening it picked. Moreover, if we set restrictions on the first several moves, we will do relative modification. For example, if only rook can be moved in a specific moves, we will set that restriction in input, so that when user input a wrong piece, we will ask the user to in put again. Also, if only a specific direction is allowed for that piece, we will check whether the coordinates the user input is the valid one.

*Question 2. How would you implement a feature that would allow a player to undo his/her last move? What ab out an unlimited number of undos?*

First let’s try to add the functionality to undo the last step:

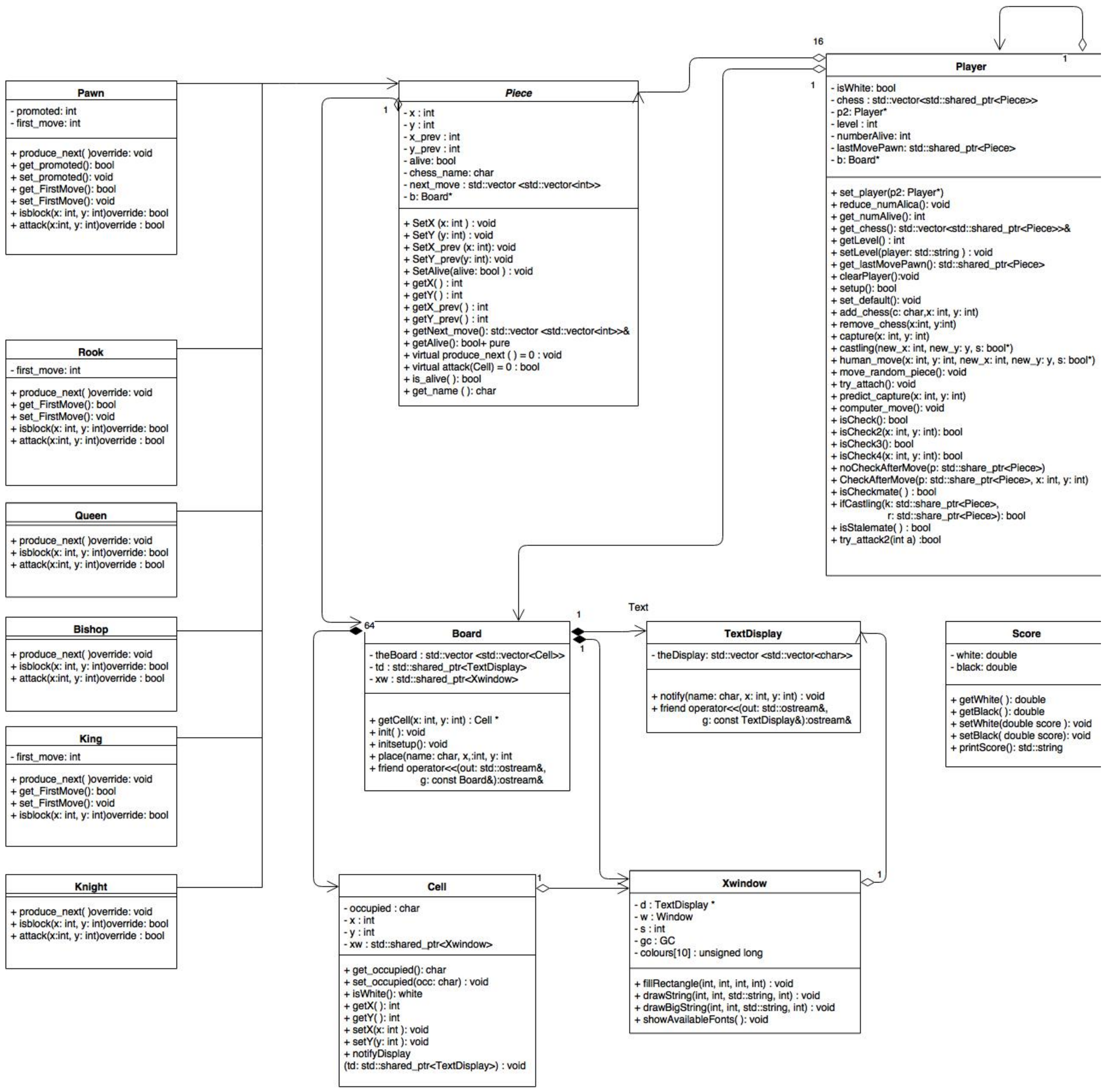
Inside Piece class, which is the mother/base class of all chess roles including queen, king … etc, it will have another 2 fields x_prev and y_prev which are used to record the last position of this piece. To be exact, when moving a chess, the current location (x and y) will be saved into x_prev and y_prev before they are updated with the new axis coordinator. For each of six roles, it will also have a field a s a flag to indicate whether it’s the last piece moved by this user. When a user hits undo feature, it will first call the corresponding player class (whether player1 or player2 hits the undo) and then check which chess has that flag on. When finding it, the current position (x and y) will be replaced with x_prev and y_prev while x_prev and y_prev can be set to - 1 to indicate we undo this chess. And the flag will be turned off in case the user wants to move another chess role.

Now let’s try to solve the issue with unlimited undo features:

Obviously, adding x_prev and y_prev won’t solve this issue. So we will need a stack to track all the movement of each chess. Exactly, there will be a vector of piece inside each player class. Every time if player1 moves a chess, we create a copy of that cell and push it into this vector before the current position x and y are changed. The same as player2. So imagine player1 now hits undo, the last element of this vector will be pop out from vector inside player1 class. We will use the function inside piece “get_name”, “get_x” and “get_y” to see which role (king or queen or ...etc) exactly is moved and what its original coordinator is. Now we can go back to the corresponding player class and find that chess role and update i ts coordinator using the one we get above from that copied cell. The same for player2 and by using this vector to keep track of all movements of two players, we can support the feature of unlimited undo.

*Question 3. Variations on chess abound. For example, four - handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four - handed chess game.*

First of all, we will modify the class Cell and Board to enlarge the board size we have right now; the TextDisplay and GraphicDisplay will also have relative changes. Then we will create four player subjects when we run the game. We add two additional setup model in which one is 2 vs 2, and another is last standing win. Players can choose the mode of the game, and the play order of will be clockwise. If it is team - up model, then the six subclasses of Piece will be modified such that players cannot check their own team members. Also the Players class will have a pointer to their teammate to identify teammates and opponents. We will modify the main function so that once a player’s king is checkmated, the player’s other pieces remain the same positions and the player will be skipped in each turn.



Pawn
- promoted: int
- first_move: int
+ produce_next( )override: void
+ get_promoted(): bool
+ set_promoted(): void
+ get_FirstMove(): bool
+ set_FirstMove(): void
+ isblock(x: int, y: int)override: bool
+ attack(x:int, y: int)override : bool
Rook
- first_move: int
+ produce_next( )override: void
+ get_FirstMove(): bool
+ set_FirstMove(): void
+ isblock(x: int, y: int)override: bool
+ attack(x:int, y: int)override : bool
Queen
+ produce_next( )override: void
+ isblock(x: int, y: int)override: bool
+ attack(x:int, y: int)override : bool
Bishop
+ produce_next( )override: void
+ isblock(x: int, y: int)override: bool
+ attack(x:int, y: int)override : bool
King
- first_move: int
+ produce_next( )override: void
+ get_FirstMove(): bool
+ set_FirstMove(): void
+ isblock(x: int, y: int)override: bool
Knight
+ produce_next( )override: void
+ isblock(x: int, y: int)override: bool
+ attack(x:int, y: int)override : bool
Piece
- x : int
- y : int
- x_prev : int
- y_prev : int
- alive: bool
- chess_name: char
- next_move : std::vector <std::vector<int>>
- b: Board*
+ SetX (x: int ) : void
+ SetY (y: int) : void
+ SetX_prev (x: int): void
+ SetY_prev(y: int): void
+ SetAlive(alive: bool ) : void
+ getX( ) : int
+ getY( ) : int
+ getX_prev( ) : int
+ getY_prev( ) : int
+ getNext_move(): std::vector <std::vector<int>>&
+ getAlive(): bool+ pure
+ virtual produce_next ( ) = 0 : void
+ virtual attack(Cell) = 0 : bool
+ is_alive( ): bool
+ get_name ( ): char
1
16
1
1
Player
- isWhite: bool
- chess : std::vector<std::shared_ptr<Piece>>
- p2: Player*
- level : int
- numberAlive: int
- lastMovePawn: std::shared_ptr<Piece>
- b: Board*
+ set_player(p2: Player*)
+ reduce_numAlica(): void
+ get_numAlive(): int
+ get_chess(): std::vector<std::shared_ptr<Piece>>&
+ getLevel() : int
+ setLevel(player: std::string ) : void
+ get_lastMovePawn(): std::shared_ptr<Piece>
+ clearPlayer():void
+ setup(): bool
+ set_default(): void
+ add_chess(c: char,x: int, y: int)
+ remove_chess(x:int, y:int)
+ capture(x: int, y: int)
+ castling(new_x: int, new_y: y, s: bool*)
+ human_move(x: int, y: int, new_x: int, new_y: y, s: bool*)
+ move_random_piece(): void
+ try_attach(): void
+ predict_capture(x: int, y: int)
+ computer_move(): void
+ isCheck(): bool
+ isCheck2(x: int, y: int): bool
+ isCheck3(): bool
+ isCheck4(x: int, y: int): bool
+ noCheckAfterMove(p: std::share_ptr<Piece>)
+ CheckAfterMove(p: std::share_ptr<Piece>, x: int, y: int)
+ isCheckmate( ) : bool
+ ifCastling(k: std::share_ptr<Piece>,
r: std::share_ptr<Piece>): bool
+ isStalemate( ) : bool
+ try_attack2(int a) :bool
64
1
1
Text
Board
- theBoard : std::vector <std::vector<Cell>>
- td : std::shared_ptr<TextDisplay>
- xw : std::shared_ptr<Xwindow>
+ getCell(x: int, y: int) : Cell *
+ init( ): void
+ initsetup(): void
+ place(name: char, x,:int, y: int
+ friend operator<<(out: std::ostream&,
g: const Board&):ostream&
TextDisplay
- theDisplay: std::vector <std::vector<char>>
+ notify(name: char, x: int, y: int) : void
+ friend operator<<(out: std::ostream&,
g: const TextDisplay&):ostream&
Score
- white: double
- black: double
+ getWhite( ): double
+ getBlack( ): double
+ setWhite(double score ): void
+ setBlack( double score): void
+ printScore(): std::string
1
Cell
- occupied : char
- x : int
- y : int
- xw : std::shared_ptr<Xwindow>
+ get_occupied(): char
+ set_occupied(occ: char) : void
+ isWhite(): white
+ getX( ): int
+ getY( ): int
+ setX(x: int ): void
+ setY(y: int ): void
+ notifyDisplay
(td: std::shared_ptr<TextDisplay>) : void
1
Xwindow
- d : TextDisplay *
- w : Window
- s : int
- gc : GC
- colours[10] : unsigned long
+ fillRectangle(int, int, int, int) : void
+ drawString(int, int, std::string, int) : void
+ drawBigString(int, int, std::string, int) : void
+ showAvailableFonts( ): void